Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Functional un|unparsing

Kenichi Asai · Oleg Kiselyov ·
Chung-chieh Shan

the date of receipt and acceptance should be inserted later

Abstract Danvy's *functional unparsing* problem (Danvy 1998) is to implement a type-safe 'printf' function, which converts a sequence of heterogeneous arguments to a string according to a given format. The dual problem is to implement a type-safe 'scanf' function, which extracts a sequence of heterogeneous arguments from a string by *interpreting* (Friedman and Wand 1984, 2008) the same format as an equally heterogeneous sequence of patterns that binds zero or more variables. We *derive* multiple solutions to both problems (Wand 1980b) from their formal specifications (Wand 1982b).

On one hand, our solutions show how the Hindley-Milner type system, unextended, permits accessing heterogeneous sequences with the static assurance of type safety. On the other hand, our solutions demonstrate the use of *control operators* (Felleisen et al. 1988; Meyer and Wand 1985; Wand 1985) to communicate with formats as *coroutines* (Haynes et al. 1984; Wand 1980a).

1 Introduction

Most programming languages provide a facility like `printf` and `scanf` for formatted input/output. For example, in C we write

```
printf("%d-th character after %c is %c", 5, 'a', 'f');
```

to produce the output

```
5-th character after a is f
```

according to the given *format descriptor*, the first argument of `printf`. The descriptor includes text such as `-th character after` to output verbatim, and conversion descriptors such as `%d` to specify that the corresponding argument of `printf` must be an integer and it should be converted to a decimal string and written out. Dually, the function `scanf` parses the input according to the descriptor:

```
scanf("%d-th character after %c is %c", &i, &c1, &c2);
```

Kenichi Asai
Ochanomizu University E-mail: asai@is.ocha.ac.jp

Oleg Kiselyov E-mail: oleg@okmij.org · Chung-chieh Shan E-mail: ccshan@cs.rutgers.edu

The literal text in the descriptor must match the input exactly; a formatting directive such as `%d` specifies that the input must contain a decimal string, which is parsed into a machine integer and stored in a reference cell given as the corresponding argument of `scanf`. Modulo the fact that this parsing may fail, the functions `printf` and `scanf` are dual: directed by the same descriptor, `printf` formats a sequence of values into a string, whereas `scanf` parses a string into a sequence of values. These functions come in several versions that differ in whether they write to and read from the console, a file, or a string. Because input/output is not the topic of this paper, we consider only the `printf` that returns a string and the `scanf` that parses a string argument (sometimes called `sprintf` and `sscanf`).

As shown above, the number and types of the values formatted by `printf` and produced by `scanf` vary depending on the format descriptor. For example, the descriptor above dictates that `printf` must receive three arguments, an integer and two characters in that order. In general, descriptors are just strings and can be built dynamically rather than specified literally. Thus, if we want the compiler to assure that `printf` receives the right number and types of arguments, then dependent types seem required. That is why most languages with `printf` do not provide such a static assurance, even though mismatches lead to wrong outputs, exceptions, and program crashes. (C compilers often warn of mismatches but cannot provide any reliable assurance.) OCaml is one of the few systems that statically detect mismatches – by extending the Hindley-Milner type system with custom rules. The main drawback of this ad-hoc approach is its limited extensibility: programmers cannot introduce conversion descriptors for their own data types, or write a new version of `printf` that sends its output to a different place.

It turns out that `printf` can be expressed with static type-checking in the unextended Hindley-Milner type system (Danvy 1998). In this paper, we *derive* that and other implementations of `printf` as well as of `scanf` starting with their formal specifications. We reproduce well-known implementations and derive several novel ones. All our implementations statically ensure that the types and the number of items to format or parse match the format descriptor. The descriptors used by `printf` and `scanf` share the same structure. Inspired by Mitchell Wand's work (Friedman and Wand 1984, 2008; Kohlbecker and Wand 1987; Wand 1980b, 1982b), our derivation repeatedly takes advantage of different representations of the same abstract object. In particular, we treat both value sequences and format descriptors symmetrically as heterogeneous sequences, and apply program transformations to fuse them with their contexts of use so as to make them palatable to the type system.

Figure 1 maps out our implementations and derivation steps. The structure of the paper is as follows.

- §2 specifies the problem formally, in the now common style of Wand (1982b), as a set of equations whose non-logical symbols are the names of functions to be implemented.
- §3 derives implementations of `printf` and `scanf` that operate on tuples rather than strings or successive arguments. The key insight is to regard `printf` and `scanf` as interpreters (Friedman and Wand 2008) of the domain-specific language of format descriptors, then to implement them as a final algebra (Kamin 1983; Wand 1979).
- §4 further transforms the implementation of `scanf` to match OCaml's curried interface. We conduct the transformations by postulating helpful properties and then

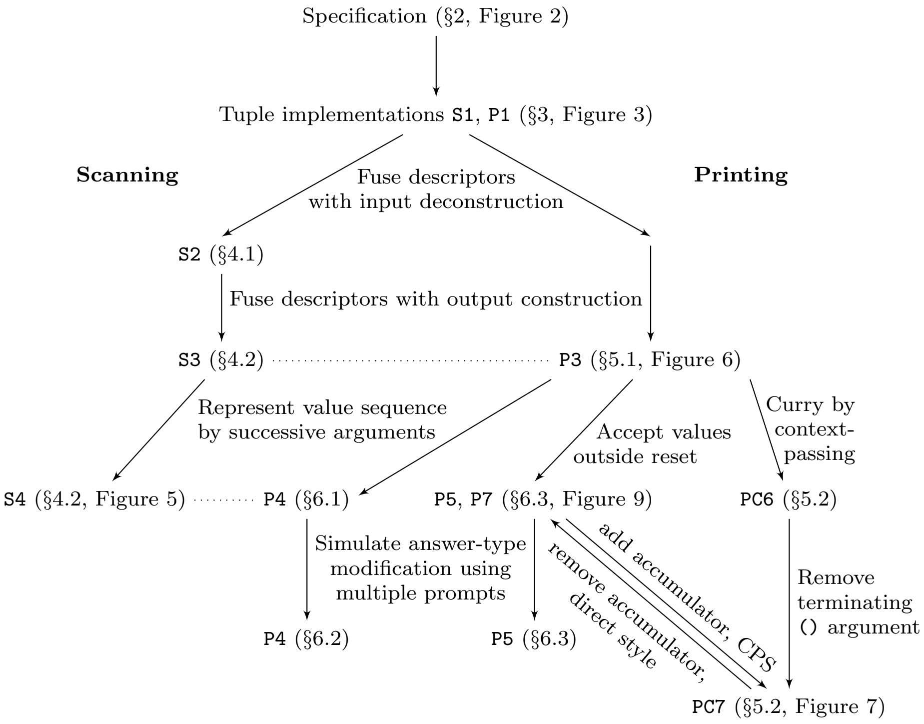


Fig. 1 Map of this paper. Nodes are implementations; edges are derivation steps.

choosing concrete representations to fit the properties – the approach elucidated by Wand (1982a) and Kohlbecker and Wand (1987).

- §5 transforms the implementation of `printf` similarly and yields code in continuation-passing style (CPS). We will introduce accumulators, which play the role of "data structure continuations" (Wand 1980b) (whose abstract, representation-independent form is described by Felleisen et al. (1988)). We also rely extensively on a transformation that fuses an expressions with a part of its context, or continuation. "By considering continuations, local transformation strategies can take advantage of global knowledge" (Wand 1980b).
- §6 uses delimited control operators to convert the implementation of `printf` to direct style. We still rely on continuation-based transformation strategies (Wand 1980b). However, we keep continuations implicit. Expressions that need to manipulate their context use control operators to obtain their continuation in the reified (Friedman and Wand 1984) form.
- §7 discusses more related work.

Our OCaml code is all available online at `http://okmij.org/ftp/typed-formatting/`.

2 Specification

We begin by formally stating the specifications from which to derive implementations. Our specifications consist of equations that mention the procedures to be implemented as undefined terms (Wand 1982b), namely `printf` and `scanf` along with constructors of heterogeneous sequences. The utility of this approach is attested by the ease with which these equations serve both as a user interface and as a starting point for implementation.

2.1 Format descriptor

Both `printf` and `scanf` receive a format descriptor as their first argument. The format descriptor is a sequence of *primitive descriptors*. We may omit the qualification 'primitive' if no confusion arises. For clarity, we limit ourselves to three primitive descriptors:

- `lit "str"`, to instruct `printf` to place the literal string `str` into the output, and to instruct `scanf` to skip `str` in the input;
- `char`, to put a character into the formatted output or to read a character from the input; and
- `int`, to format an integer as a decimal string, or to parse the input as a decimal string.

We shall see that in all our implementations the set of primitive descriptors is user-extensible.

To arrange primitive descriptors into a sequence, we write `nilD` for the empty sequence and `consD h t` for the sequence whose first element is `h` and rest is `t`. In our running example, the C format descriptor `"%d-th character after %c is %c"` can be written as follows:

```
consD int
 (consD (lit "-th character after ")
   (consD char (consD (lit " is ") (consD char nilD))))
```

We abbreviate this sequence like an OCaml list:

$$\texttt{[int; lit "-th character after "; char; lit " is "; char]}_D$$

The subscript by the right bracket reminds us that this is not a list but a heterogeneous sequence.

At present, primitive descriptors like `int` and the sequence constructors `nilD` and `consD` are abstract. As we derive implementations in the later sections, the concrete realizations of these constructors will suggest themselves. At that point we will consider typing, which we leave aside in this section.

Format descriptors constitute a simple domain-specific language (DSL), whose phrases (terms) are built by attaching (with `consD`) primitive descriptors to `nilD`. This embedded DSL lets us write grammars of `printf`'s output or `scanf`'s input, in the form of OCaml expressions. The expressions can then be interpreted as parsers or pretty-printers for these grammars. One may therefore regard our format descriptors as parser/pretty-printer combinators – albeit quite simple ones. Unlike the full-fledged parser combinator frameworks (Swierstra 2009), we provide only sequential composition, with no choice or recursion.

2.2 Printf

Given a format descriptor, `printf` receives a variable number of arguments of various types, and produces a string. The number and types of the arguments must match the format descriptor. For example,

```
   printf [int; lit "-th character after "; char; lit " is "; char]_D
          5 'a' 'f'
 = "5-th character after a is f"
```

To ease specification and reasoning, and to highlight the symmetry between `printf` and `scanf`, we treat the arguments to `printf` and the resulting string as sequences, too:

```
   printf [int; lit "-th character after "; char; lit " is "; char]_D
          [5; (); 'a'; (); 'f']_A
 = ["5"; "-th character after "; "a"; " is "; "f"]_S
```

We made three changes, to be reverted in the derivation in §5:

- First, the arguments are given abstractly as a heterogeneous sequence, built with the constructors `nilA` and `consA`.
- We have also changed the behavior of `lit`. To format a character or an integer, we have to give the value to format in the argument sequence. In the case of `lit`, the literal string to output is given in the descriptor itself and so there is no need for a corresponding element in the argument sequence. Nevertheless, for uniformity we do require an argument `()`.
- Finally, the output is returned abstractly as a sequence of strings, built using the constructors `nilS` and `consS`, rather than a single string.

We are ready to specify `printf`. Assume that `string_of_char c` and `string_of_int i` convert a character `c` and an integer `i` to a string, respectively. (Below, they are abbreviated as `s_of_c c` and `s_of_i i` to fit the definition in each line.)

```
printf nilD                  nilA           = nilS
printf (consD (lit s) desc) (consA () lst) = consS s          (printf desc lst)
printf (consD int     desc) (consA i  lst) = consS (s_of_i i) (printf desc lst)
printf (consD char    desc) (consA c  lst) = consS (s_of_c c) (printf desc lst)
```

This specification is not executable code. We have yet to implement constructors such as `consD`, `nilD`, and `consA`, which are not necessarily data constructors, and to decide what it means to pattern-match on them. We also ignore typing until we fix the representation of heterogeneous sequences.

If we regard format descriptors as a DSL, then `printf` is one interpreter of the language, interpreting each phrase (term) as a function from an argument sequence to a string sequence.

2.3 Scanf

We take the interface of the standard OCaml function `sscanf` as the specification for our `scanf`: Given a format descriptor, `scanf` receives the input string and a consumer function accepting the parsed values. The number and types of the arguments of the consumer function must match the format descriptor. The result of the consumer function is returned as the result of `scanf`. For example,

```
   scanf [int; lit "-th character after "; char; lit " is "; char]_D
         "5-th character after a is f"
         f
 = f 5 'a' 'f'
```

Following `printf`, we revise this behavior to a more uniform one:

```
   scanf [int; lit "-th character after "; char; lit " is "; char]_D
         ["5"; "-th character after "; "a"; " is "; "f"]_S
 = [5; (); 'a'; (); 'f']_A
```

In addition to the three changes made for `printf`, we removed the consumer function and made `scanf` return an abstract sequence instead. In §4, we re-introduce the consumer function and resolve the problem of breaking the input string into a sequence of tokens such as `"5"`, `"a"`, etc.

We specify the behavior of `scanf` as follows. Assume that `char_of_string s` (abbreviated as `c_of_s s`) converts a string `s` of length one into a character and `int_of_string s` (abbreviated as `i_of_s s`) converts a string `s` into an integer.

```
scanf nilD                nilS          = nilA
scanf (consD (lit s) desc) (consS s lst) = consA ()         (scanf desc lst)
scanf (consD int     desc) (consS s lst) = consA (i_of_s s) (scanf desc lst)
scanf (consD char    desc) (consS s lst) = consA (c_of_s s) (scanf desc lst)
```

We see that `scanf` is also an interpreter for the DSL of format descriptors. Dually to `printf`, `scanf` interprets each phrase (term) as a function from a string sequence to a result sequence.

2.4 Uniform specification

The specification for `printf` in Section 2.2 is uniform: all cases except for `nilD` look very much alike. We can condense them by introducing a generic primitive descriptor `dP` as a function that takes a value and converts it to a string. We can re-write the specification of `printf` in two lines:

```
printf nilD nilA = nilS
printf (consD dP desc) (consA x lst) = consS (dP x) (printf desc lst)
```

Figure 2 introduces the primitive descriptors `litP s`, `intP` and `charP` to which `dP` could be instantiated. Now we define these primitive descriptors not abstractly but as concrete OCaml code. The same generalization applies to `scanf`: we introduce a generic format descriptor `dS` that can be one of `litS s`, `intS` and `charS`. In `litS`, the check `assert (s = s')` assures that the input string `s'` is exactly the same as specified by `litS`. Clearly the set of primitive descriptors is extensible: we can easily add `floatP` and `floatS` for formatting floating-point numbers by analogy with `intP` and `intS`. In fact, we can add primitive descriptors for any object *obj* provided we can write functions `string_of_`*obj* and *obj*`_of_string`. Furthermore, we can parameterize the descriptors, for example, by giving to `intS` and `intP` additional arguments specifying padding, field width, etc.

The condensed specification for `printf` and `scanf` at the top of Figure 2 exhibits a pleasant symmetry. One may even think that `printf` and `scanf` are the inverses of each other. However, that is not quite right. For example, `int_of_string` used in the primitive descriptor `intS` may strip leading zeroes when reading a number; the corresponding `string_of_int` used in `intP` prints numbers without leading zeroes. Furthermore,

`scanf [int; char]`$_D$ `(printf [int; char]`$_D$ `[5; '4']`$_A$`)`

reproduces the original sequence `[5; '4']`$_A$ only if the result of `printf` is kept segmented or is appropriately tokenized when given to `scanf`. Otherwise, `scanf` will fail because the format descriptor `int` has read ‘too much’ (we return to this issue in §4). Therefore, we require `scanf` and `printf` to be weak inverses of each other:

```
printf descP (scanf descS (printf descP arg)) = (printf descP arg)
scanf descS  (printf descP (scanf descS str)) = (scanf descS str)
```

provided that the innermost `scanf` expressions on the left-hand-side do yield a result. The notation `descS` means `descP` with primitive descriptors `intP` replaced with `intS`, etc.

So far we have shown only the specification. Deriving the actual code is the subject of the following sections.

3 Deriving the tupling implementation

The goal of this section is to transform the specification of `printf` and `scanf` into typable code in a rigorous manner, selecting concrete representations for the abstract sequences. In this section, we represent the argument and string sequences simply as nested tuples. The key idea is to view the specification of `printf` and `scanf` as defining two interpreters (Friedman and Wand 2008) of a domain-specific language of format descriptors.

First we have to choose the representation for `nilD` and `consD`. The most obvious choice – to identify them with the constructors `[]` and `(::)` of ordinary OCaml lists – is problematic. The primitive descriptors such as `charP` and `intP` have different types and cannot be put into the same ordinary list. The corresponding elements of the argument sequence may also be of different types. Furthermore, we would like inconsistent formatting expressions such as

```
printf (consD int nilD) (consA 'c' nilA)
```

to be rejected by the type-checker as ill-typed. Therefore, we must avoid the universal type and keep the sequences of descriptors and of arguments heterogeneous.

The view of format descriptors as a DSL offers the needed insight. To be able to statically reject `printf` expressions where the number or the types of elements in the argument sequence do not match the format descriptor, we should make our DSL typed and its interpreter `printf` type-preserving. We can implement such a typed DSL and type-preserving interpreter in two ways – a *deep* or *initial* embedding (Goguen et al. 1978), or a *shallow* or *final* embedding (Hudak 1996; Kamin 1983; Wand 1979).

Main functions:

```
printf nilD nilA = nilS
printf (consD dP desc) (consA x lst) = consS (dP x) (printf desc lst)

scanf nilD nilS = nilA
scanf (consD dS desc) (consS s lst) = consA (dS s) (scanf desc lst)
```

Format directives `dP` for `printf`:

```
(* litP : string -> unit -> string *)
let litP s = fun () -> s

(* intP : int -> string *)
let intP   = fun i  -> string_of_int i

(* charP : char -> string *)
let charP  = fun c  -> string_of_char c
```

Format directives `dS` for `scanf`:

```
(* litS : string -> string -> unit *)
let litS s = fun s' -> assert (s = s'); ()

(* intS : string -> int *)
let intS   = fun s -> int_of_string s

(* charS : string -> char *)
let charS  = fun s -> char_of_string s
```

Expected behavior:

```
  printf [intP; litP "-th character after "; charP; litP " is "; charP]_D
         [5; (); 'a'; (); 'f']_A
= ["5"; "-th character after "; "a"; " is "; "f"]_S
  scanf [intS; litS "-th character after "; charS; litS " is "; charS]_D
         ["5"; "-th character after "; "a"; " is "; "f"]_S
= [5; (); 'a'; (); 'f']_A
```

Fig. 2 The uniform specification

The initial embedding relies on a *generalized* algebraic data type (GADT) (Xi et al. 2003). We instead use the final embedding, which can be easily implemented in OCaml (Carette et al. 2009).

The final embedding implements a typed interpreter as a fold over DSL terms. At first blush, `printf` does not seem to be a fold. Rather, it is `zipWith` (with the restriction that two lists must have the same size). The function `zipWith` has the following specification.

```
zipWith f nil nil = nil
zipWith f (cons x l1) (cons y l2) = cons (f x y) (zipWith f l1 l2)
```

(where `nil` and `cons` are abstract sequence constructors). Comparing to Figure 2, we see that both `printf` and `scanf` are instances of `zipWith` whose first argument `f` is function application.

However, we can easily transform `zipWith` into a fold. First, we re-write the above specification of `zipWith` as follows:

```
zipWith f nil = fun nil -> nil
zipWith f (cons x l1) = fun (cons y l2) ->
                          cons (f x y) ((zipWith f l1) l2)
```

(implicitly relying on some sort of pattern-matching in the argument of the function. We fix the exact form of that later.) We re-write further as:

```
zipWith f nil = fun nil -> nil
zipWith f (cons x l1) = (fun x -> fun t ->
                          fun (cons y l2) -> cons (f x y) (t l2))
                        x (zipWith f l1)
```

Comparing this expression with the specification for the `fold`

```
fold g z nil         = z
fold g z (cons x l1) = g x (fold g z l1)
```

we see that `zipWith` is an instance of `fold`:

```
zipWith f = fold g z
where g x t = fun (cons y l2) -> cons (f x y) (t l2)
      z     = fun nil -> nil
```

That conclusion is formally justified by the universality of `fold` (Hutton 1999). Thus we can re-write the specification of `printf` (Figure 2 or §2.4) in terms of `fold`:

```
printf desc args = fold g z desc args
where g dP tD = fun (consA y l2) -> consS (dP y) (tD l2)
      z       = fun nilA -> nilS
```

The reformulation using `fold` gives us insight into a well-typed implementation. Let us take an example of applying `printf` to a sample descriptor sequence:

```
  printf (consD int (consD char nilD))
= {inline the definition of printf}
  fold g z (consD int (consD char nilD))
= {apply the equational specification of fold}
  (g int (g char z))
```

Informally, the net effect of `fold` is indeed replacing the sequence constructors `consD` and `nilD` with `g` and `z` (Hutton 1999). Recall that `consD` and `nilD` have deliberately been left unspecified, appearing in the specification as free identifiers. We can avoid the building of the sequence using some `consD` and `nilD` and the subsequent replacement of these constructors with `g` and `z` if we choose `consD` as `g` and `nilD` as `z` to begin with. Our sample descriptor sequence `[int; char]`$_D$ becomes `(g int (g char z))`, which is identical to the result of the `printf` application on the last line in the derivation above. With this choice for the descriptor sequence constructors, `printf` becomes the identity. In other words, we have applied deforestation (Wadler 1990) to replace intermediate trees, namely format descriptors, by their `printf` interpretation. Since we no longer have to build these intermediate trees, we no longer have any typing problems of building heterogeneous data structure. Carette et al. (2009) elaborate on using similar deforestation to resolve the typing problems in representing DSL expressions.

By choosing the tuple-encoding for the argument sequence (that is, setting `nilA` to be `()` and `consA x y` to be `(x,y)` so that we can pattern-match on `nilA` and `consA x y`), we obtain

```
let consD dP tD = fun (y,l2) -> consS (dP y) (tD l2)
let nilD        = fun () -> nilS
```

We can choose the nested tuple representation for the output sequence as well. This gives us the real, well-typed OCaml implementation for `printf`, summarized in Figure 3. Our running example can now be written in OCaml. The format descriptor (which we bind to the identifier `descP1` for ease of reference) becomes

```
let descP1 = consD intP
               (consD (litP "-th character after ")
                 (consD charP
                   (consD (litP " is ")
                     (consD charP
                       nilD))))
```

We use the descriptor to format a sample sequence of arguments (which we bind to the identifier `arg1` for ease of reference)

```
let arg1 = (5,((),('a',((),('f',())))))
```

obtaining the result

```
let r1 = printf descP1 arg1
```

which OCaml prints as

```
val r1 : string * (string * (string * (string * (string * unit)))) =
  ("5", ("-th character after ", ("a", (" is ", ("f", ())))))
```

We emphasize the type of `descP1`, which OCaml infers to be

```
int * (unit * (char * (unit * (char * unit)))) ->
  string * (string * (string * (string * (string * unit))))
```

It is a function from a typed heterogeneous sequence of arguments to a sequence of strings.

We can also choose `nilS` to be the empty string and `consS` to be the string concatenation, so that the formatting result becomes the familiar string.

The case for `scanf` is nearly identical and so we show only the final result of the derivation, the OCaml implementation in Figure 3. The format descriptor for `scanf` is built with the same sequence constructors `nilD` and `consD` but using different primitive descriptors: `intS` rather than `intP`, etc. For example, the format descriptor for our running `scanf` example is

```
let descS1 = consD intS
               (consD (litS "-th character after ")
                 (consD charS
                   (consD (litS " is ")
                     (consD charS
                       nilD))))
```

whose inferred type is

```
string * (string * (string * (string * (string * unit)))) ->
  int * (unit * (char * (unit * (char * unit))))
```

According to the type, `descS1` is the inverse of `descP1`. Indeed, feeding the result of the format output (which we bound to `r1` earlier) to the format input

Main functions:

```
let printf = fun x -> x
let scanf  = fun x -> x
```

Constructors for the format descriptor sequence:

```
(* nilD : unit -> unit *)
let nilD = fun () -> ()

(* val consD : ('a -> 'b) -> ('c -> 'd) -> 'a * 'c -> 'b * 'd *)
let consD d tD = fun (y, l2) -> ((d y),(tD l2))
```

Constructors for the argument sequence:

```
let nilA = ()
let consA x y = (x,y)
```

Constructors for the result sequence:

```
let nilS = ()
let consS x y = (x,y)
```

Fig. 3 The tuple implementation. Primitive descriptors are implemented in Figure 2.

```
scanf descS1 r1
```

gives us

```
- : int * (unit * (char * (unit * (char * unit)))) =
(5, ((), ('a', ((), ('f', ())))))
```

which is the argument sequence `arg1` used for producing the format output.

4 Typed `scanf`

In the previous section, we derived one implementation of typed `printf` and `scanf`, achieving our goal of reporting the mismatch between the data to format and the format descriptor as a type error. However, we fell short of implementing the interface of OCaml's built-in formatted-IO functions. In particular, we would like `printf` to receive the data to format as successive curried function arguments rather than as a nested tuple. We would like `scanf` to pass the parsed data to a consumer function rather than returning them as a nested tuple. What's more, we would like `scanf` to take as input a string or a stream rather than an already appropriately tokenized sequence.

In this section, we adjust the earlier derivation of `scanf` and attain our desideratum: an implementation of `scanf` that matches the interface of OCaml's built-in formatted input. We achieve this goal by instantiating the abstract sequences in various ways, not just as nested tuples to pattern-match against, but also as abstract types equipped with deconstructor functions. We follow the pattern demonstrated in (Kohlbecker and Wand 1987; Wand 1982a) of postulating properties (equational laws) that help transform specifications in desirable ways, and then choosing concrete representations to fit the properties. We exploit the freedom of choice for the representation to derive several implementations of `scanf`, eventually attaining our desideratum. We deal with `printf` in the next section.

We recall the general implementation of `nilD` and `consD` for `scanf`:

```
let nilD       = fun nilS -> nilA
let consD d tD = fun (consS y l2) -> consA (d y) (tD l2)
```

The `scanf` of the previous section was obtained as we set `nilS` and `nilA` to be `()` and `consA` and `consS` to be the tuple constructor. We will make a different choice in this section.

4.1 Moving input deconstruction into primitive format descriptors

We start by examining the pattern-matching in the definition for `nilD` and `consD` above. The pattern-match in `fun nilS -> nilA` means checking if the argument of the function matches `nilS`: in other words, if the input is finished. Unconsumed input signifies the inconsistency with the format descriptor `nilD`. We can also write this check as `fun x -> is_nilS x; nilA`, assuming an assertion `is_nilS` that checks if the input is finished, raising a compile- or a run-time error otherwise. Similarly, we can re-write the pattern-match in the definition of `consD` using a function `un_consS` that checks to see that the input is not finished, returning the current item along with the remainder; see Figure 4.

Using deconstructors to define `nilD` and `consD`

```
let nilD       = fun x -> is_nilS x; nilA
let consD d tD = fun x -> let (y,l2) = un_consS x in
                          consA (d y) (tD l2)
```

Properties of deconstructors

```
is_nilS nilS = ()
un_consS (consS y l) = (y,l)
```

Fig. 4 Sequence deconstructors: `is_nilS` and `un_consS`

So far, we merely restated the earlier definition for `nilD` and `consD`, using deconstructor functions rather than pattern-matching. The deconstructor functions must satisfy the specification expected of pattern-matching, that is, `is_nilS x` succeeds if and only if `x` is the empty sequence `nilS`; for any `y` and the sequence `l` so that `consS y l` (is well-typed and) terminates, `un_consS (consS y l)` (is well-typed and) terminates with the value `(y,l)`.

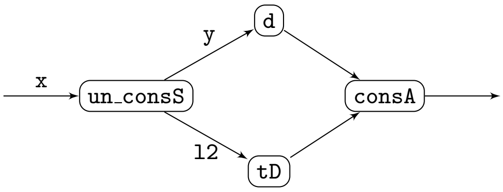
If the sequence of strings is the nested tuple, as was the case for Figure 3, then the deconstructors are

```
let is_nilS () = ()
let un_consS (h,t) = (h,t)
```

They clearly satisfy the expected properties (Figure 4): e.g., `is_nilS x` succeeds if and only if `x` is the empty sequence `nilS`, that is, `()`. In fact, `is_nilS x` is not even well-typed if `x` is not `()`. Substituting these deconstructors in the definition for `nilD` and `consD` in Figure 4, we recover the implementation in Figure 3.

The introduction of deconstructors offers a different perspective on `nilD` and `consD`. We may view the sequence of strings as an “input stream”. The function `is_nilS` checks that the stream is finished. The function `un_consS` “reads” from the (non-empty) stream, returning the pair of the current element and the rest of the stream.

The operation `consD d tD` as defined in Figure 4 could be understood as reading from the stream `x`, passing the read string value `y` to the primitive descriptor `d` for parsing, processing the rest of the input, and building the output using `consA`. The following diagram visualizes this data flow.



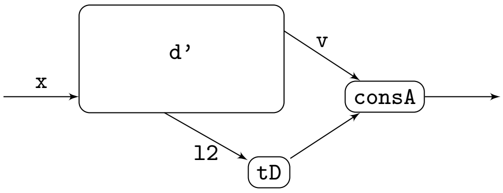
In this definition of `consD`, the input token `y` extracted by `un_consS` is used only once: it is passed to the primitive descriptor `d`. We see a chance to fuse the deconstruction of the input with the parsing of the token:

```
let consD d’ tD = fun x -> let (v,l2) = d’ x in
                           consA v (tD l2)
```

That is, we re-define the primitive descriptors (the element of the descriptor sequence), notated as `d’` above so they “read” from the input stream themselves. To be precise, the primitive format descriptors for `scanf` are to take the input sequence, parse the current item and, if successful, return the parsing result along with the remainder of the input sequence:

```
let litS s = fun x ->
  let (y,l2) = un_consS x in
  assert (s = y); ((),l2)
let intS  = fun x ->
  let (y,l2) = un_consS x in
  (int_of_string y,l2)
let charS = fun x ->
  let (y,l2) = un_consS x in
  (y.[0],l2)
```

Compared with the earlier definition in Figure 2, the new format descriptors invoke `un_consS` themselves. The significance of the new definitions is that the reading of the input stream is no longer done generically by `consD`. Rather, each primitive descriptor such as `litS` reads the input stream in its own way.



When the input sequence is a nested tuple of string fragments, it is not so useful for primitive descriptors to read and parse the input sequence in their own ways. To actually use this new flexibility, we represent the input sequence differently. Instead of a nested tuple, we take it to be a single string:

```
let nilS  = ""
let consS = (^)
let is_nilS x = assert (x = "")
```

The empty sequence is the empty string and `consS` is string concatenation. The deconstructor `is_nilS` verifies that its argument is the empty string. The deconstructor `un_consS` splits a non-empty string into a prefix and a suffix. Since the splitting can be done in many ways, the deconstructor `un_consS` is generally a relation rather than a function. Therefore, the second property in Figure 4 has to be relaxed: `un_consS` relates `consS y l` to `(y,l)`; that is, there is *some way* to deconstruct the sequence `consS y l` to obtain the components `y` and `l`. Also, `scanf` may not always be able to parse the string produced by `printf` using the same format descriptor: the string `"12"` can be produced by formatting two numbers but should only be parsed as one number.

The fact that deconstructing the input is a relation complicates the derivation of `scanf`. This is where it helps to move `un_consS` into the primitive descriptors. Informally, we may view `un_consS` as a non-deterministic function that generates various prefixes of the input. A primitive descriptor tests the deconstruction candidates and either accepts one of them or raises a parsing error. The primitive descriptor returns the result of parsing the prefix along with the suffix of the input. For example, we may view the format descriptor `litS s` as invoking `un_consS inp` to generate deconstructions of the input string `inp` into a prefix `y` and the rest `l`. We then test if any candidate `y` is equal to the string `s`. If so, we return the parsing result, `()`, along with `l`. If not, we raise the exception `Scan_error`.

```
exception Scan_error of string
```

This generate-and-test process is easy to program deterministically: it simply checks if the input string `inp` has the prefix `s`:

```
(* val litS2 : string -> string -> unit * string *)
let litS2 str = fun inp ->
  if length str <= length inp &&
     str = sub inp 0 (length str)
  then ((), sub inp (length str) (length inp - length str))
  else raise (Scan_error "lit")
```

Throughout the paper, we develop many versions of the code, illustrating different approaches or progressive improvements. To tell the versions apart we attach a numeric suffix to the names of the functions, for example, `litS2`. The suffix also makes it easy to correlate the snippets in the paper with the complete code accompanying the paper.

We now see the benefit of moving `un_consS` into the primitive descriptors: by bringing the generation (of input prefixes) and testing closer, we can make parsing deterministic. A primitive descriptor `charS` accepts the single-character prefix of the input string, returning the character as the parsing result. The descriptor `intS` accepts the longest prefix of the input string that can still be parsed as an integer.

```
(* val intS2 : string -> int * string *)
let intS2 = ...

(* val charS2 : string -> char * string *)
let charS2 = fun inp ->
  if length inp = 0
  then raise (Scan_error "char")
  else (inp.[0], sub inp 1 (length inp - 1))
```

4.2 Returning parsing results as successive arguments

The input to `scanf` is now a string, but the result is still a nested tuple such as `arg1`. In particular, it still contains dummy values. We may be tempted to write something like the following

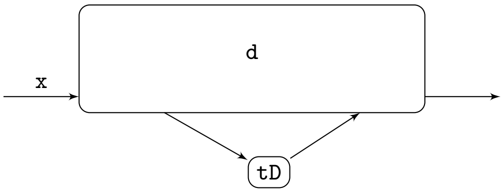
```
let consDS d tD = fun x -> let (vy,l2) = d x in
                           if vy = () then tD l2 else
                           consA vy (tD l2)
```

but that is clearly untypable because the expressions in the two branches of `if` generally have different types (for example, different nested-tuple types).

To get rid of the dummy `()` in the resulting tuple, we fuse further, letting a primitive descriptor itself construct the resulting sequence. For example, `charS2` becomes:

```
(* val charS3 : (string -> 'a) -> string -> char * 'a *)
let charS3 = fun tD inp ->
  if String.length inp = 0
  then raise (Scan_error "char")
  else consA inp.[0] (tD (String.sub inp 1 (String.length inp - 1)))
```

Previously, `charS2` returned the pair of values `(v,l2)` to be used in the expression `consA vy (tD l2)` constructing the final result. Now, we pass `charS` the remainder of the format descriptor `tD` as an argument, and let it construct the final result by invoking `consA`. The primitive descriptor now decides if to invoke `consA` or not. The following diagram illustrates this change.



The descriptor `litS`, which does not need to construct any output, would not invoke `consA`:

```
(* val litS3 : string -> (string -> 'a) -> string -> 'a *)
let litS3 str = fun tD inp ->
  if String.length str <= String.length inp &&
     str = String.sub inp 0 (String.length str)
  then tD (String.sub inp (String.length str)
                          (String.length inp - String.length str))
  else raise (Scan_error "lit")
```

We re-define `consDS` to take advantage of the new primitive descriptors:

```
(* val consDS : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c *)
let consDS d tD = fun x -> d tD x
```

Since the new primitive descriptors deconstruct the input sequence and construct the output one, the new `consD` has nothing left to do. Indeed, it is just the identity function. Our running example becomes

```
let descS3 = consDS intS3
                (consDS (litS3 "-th character after ")
                  (consDS charS3
                    (consDS (litS3 " is ")
                      (consDS charS3
                        nilDS))))
```

whose inferred type is `string -> int * (char * (char * unit))`. Here `nilDS` stands for `nilD` shown earlier. Applying the descriptor to the input string

```
descS3 "5-th character after a is f"
```

returns the result `(5, ('a', ('f', ())))`. It is still the nested tuple but has no dummy `()` (compare with `arg1`).

We still have not derived the OCaml-like `sscanf`, which should pass the parsed data to a consumer function rather than returning them in a nested tuple. Therefore, we chose a different representation for the sequence of parsed values. We re-define `nilA` and `consA` as follows:

```
let nilA = fun f -> f
let consA h t = fun f -> t (f h)
```

For motivation, consider a sample sequence `(consA x1 (consA x2 nilA))`, which is β-equivalent to `fun f -> f x1 x2` – a sequence of arguments passed to a consumer function. That is exactly how we would like to return the parsing results. Using this new definitions of `nilA` and `consA` immediately gives us the desired implementation of `scanf`, summarized in Figure 5.

Since `consDS` is the identity function, we may drop it when writing the format descriptors. We then notice that the format descriptor is the functional composition of primitive descriptors, applied to `nilDS`. The source code accompanying the paper, the file `derive5.ml`, shows the further generalization, letting us parse data from an arbitrary source (a string, a file stream, etc) while using the same format descriptor. We observe in passing that the type of `litS4 str` is `(string -> 'a) -> string -> 'a`, which is the type of the CPS transform of a `string -> string` function. The observation suggests that the argument `tD` may be regarded as a continuation.

5 Typed polyvariadic `printf`

We now turn to deriving the implementation of the OCaml-like `printf` outlined at the beginning of §2.2 from the specification at the end of that section. We proceed in several steps. We have already derived the tupling implementation, §3 and Figure 3; we now refine it into the desired implementation. Like in §4, we will be instantiating the abstract sequences in various ways. We will also perform simple program manipulations such as inlining, uncurrying, and changing the argument order for the sake of later η-reductions. An early and lucid example of such transformation-based program development is that of Kohlbecker and Wand (1987). We make extensive use of a less simple transformation: fusing an expression with a bit of its context so that local transformation strategies can take advantage of global knowledge (Wand 1980b).

Main function:

```
let scanf  = fun x -> x
```

Constructors for the format descriptor sequence:

```
(* val nilDS : string -> 'a -> 'a *)
let nilDS       = fun x -> is_nilS x; nilA

(* val consDS : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c *)
let consDS d tD = fun x -> d tD x  (* identity *)
```

Constructors for the result sequence:

```
let nilA = fun f -> f
let consA h t = fun f -> t (f h)
```

Primitive format descriptors:

```
(* val litS4 : string -> (string -> 'a) -> string -> 'a *)
let litS4 str = fun tD inp ->
  if String.length str <= String.length inp &&
     str = String.sub inp 0 (String.length str)
  then tD (String.sub inp (String.length str)
                          (String.length inp - String.length str))
  else raise (Scan_error "lit")

(* val intS4 : (string -> 'a -> 'b) -> string -> (int -> 'a) -> 'b *)
let intS4 = fun tD inp -> ...

(* val charS4 : (string -> 'a -> 'b) -> string -> (char -> 'a) -> 'b *)
let charS4 = fun tD inp ->
  if String.length inp = 0
  then raise (Scan_error "char")
  else consA inp.[0] (tD (String.sub inp 1 (String.length inp - 1)))
```

Running example:

```
let descS4 = consDS intS4
               (consDS (litS4 "-th character after ")
                 (consDS charS4
                   (consDS (litS4 " is ")
                     (consDS charS4
                       nilDS))))
(* val descS4 : string -> (int -> char -> char -> '_a) -> '_a *)

descS4 "5-th character after a is f" (fun x1 x2 x3 -> (x1,x2,x3))
(*
- : int * char * char = (5, 'a', 'f')
*)
```

Fig. 5 The final implementation of `scanf`.

5.1 Refined tupling implementation

Recall that the specifications of `printf` and `scanf` are very much alike: essentially, we obtain one from the other by swapping `consA` with `consS` and `nilA` with `nilS` in Figure 2. We can thus reuse the derivation for `scanf` in §4 *mutatis mutandis*. Since the derivation has already been explained, here we will be brief. First, we write the expressions for `nilD` and `consD` using the deconstructor functions:

```
let nilD       = fun x -> is_nilA x; nilS
let consD d tD = fun x -> let (y,l2) = un_consA x in
                          consS (d y) (tD l2)
```

Compared to the same expressions at the beginning of §4, we have swapped `consA` and `consS` and replaced `un_consS` with `un_consA`. As we explained before, we move the deconstruction of the argument sequence `un_consA` and the construction of the sequence of strings `consS` into the primitive format descriptors. This immediately gives us the implementation in Figure 6. As in §4, we represent the abstract sequence of strings as a single string. The result of `printf` is now a string as desired. The arguments to `printf` are still given as a nested tuple – which, however, no longer has dummy `()` for the `lit` primitive descriptor, since `litP3` does not call `un_consA`. As was the case with the final `scanf` implementation, `consD` is just the identity, hence the format descriptor is a functional composition of primitive descriptors, applied to `nilD`.

5.2 Polyvariadic `printf`, context-passing style

In this section, we transform the refined tupling implementation in Figure 6 to `printf` that accepts values to format as function arguments rather than grouped into a nested tuple. We effectively "curry" the implementation in Figure 6. The end result of our derivation is essentially the result by Danvy (1998).

In the implementation of Figure 6, the descriptor sequence has the type `('a * ('b * ...)) -> string`, built from the primitive descriptors like

```
let intP3 = fun tD (n,l2) ->
  consS (string_of_int n) (tD l2)
```

using the combining operator `consDP` (which is just the identity). Instead, we want the descriptor sequence to have the curried type `'a -> 'b -> ... -> string`. To obtain this type, we need to curry `fun (n,l2) -> ...` in `intP3` above, so that `n` and `l2` become separate successive arguments, then somehow η-reduce `l2` away because `l2` represents an unknown number of successive arguments.

Let us try to blindly curry `intP3`, obtaining

```
let intP3' = fun tD n l2 ->
  consS (string_of_int n) (tD l2)
```

which is however problematic. Let us consider a sample sequence `consDP intP3' (consDP intP3' nilDP)`, which is `intP3' (intP3' nilDP)`. The function `nilDP` has the type `unit -> string`. The inner occurrence of `intP3'` therefore has the type `(unit -> string) -> int -> unit -> string`. The outer occurrence of `intP3'` should therefore have the type of the form `(int -> unit -> string) -> int -> ??? -> string`, leading to a contradiction. The inner `intP3'` passed its argument `tD` one value, `unit`, obtaining a `string`. The outer `intP3'` now has to pass its argument `tD` two values to get a string. Clearly we cannot write a polymorphic `intP3'` that can be instantiated to the two occurrences required by our example.

Our failed experiment showed that the curried `intP3` should pass its argument `tD` a variable number of arguments to obtain a string. We cannot write such a function using the regular parametric polymorphism of the Hindley-Milner type system. Thus the new `intP3` should not try to obtain the intermediate formatting result from `tD`, to which to prepend `string_of_int n`, the result of the formatting of its own argument.

Main function:
```
let printf  = fun x -> x
```

Constructors for the format descriptor sequence:
```
(* val nilDP : unit -> string *)
let nilDP       = fun x -> is_nilA x; nilS

(* val consDP : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c *)
let consDP d tD = fun x -> d tD x  (* identity *)
```

Deconstructors for the argument sequence:
```
let is_nilA = fun () -> ()
let un_consA x = x
```

Constructors for the result sequence:
```
let nilS = ""
let consS h t = h ^ t
```

Primitive format descriptors:
```
(* val litP3 : string -> ('a -> string) -> 'a -> string *)
let litP3 str = fun tD arg ->
  consS str (tD arg)

(* val intP3 : ('a -> string) -> int * 'a -> string *)
let intP3 = fun tD arg ->
  let (n,l2) = un_consA arg in
  consS (string_of_int n) (tD l2)

(* val charP3 : ('a -> string) -> char * 'a -> string *)
let charP3 = fun tD arg ->
  let (c,l2) = un_consA arg in
  consS (string_of_char c) (tD l2)
```

Running example:
```
let descP3 = consDP intP3
               (consDP (litP3 "-th character after ")
                 (consDP charP3
                   (consDP (litP3 " is ")
                     (consDP charP3
                       nilDP))))
(* int * (char * (char * unit)) -> string *)

descP3 (5,('a',('f',())))
(*
- : string = "5-th character after a is f"
*)
```

Fig. 6 The refined tupling implementation of `printf`.

Instead of *asking* `tD` to produce a string, the new `intP3` should *tell* `tD` to prepend `string_of_int n` to `tD`'s result after it is finally computed. Thus the new `intP3` should be written in the accumulator-passing style for the result of the formatting.

Now that we know the general form of the desired primitive descriptors, we can derive them, by a sequence of two transformations. First, we choose a slightly different representation for the string sequence. It is no longer a string. Rather, it is a function that receives a string (an accumulator) and appends to it. In short, the S-sequence is now a difference string:

```
let nilSC4       = fun acc -> acc
let consSC4 h t = fun acc -> t (acc ^ h)
```

(We shall use the version suffix `C4`, `C5`, etc. to emphasize that we are developing context-passing implementations.) Using the accumulator opens up new optimizations, such as accumulating string fragments and delaying or avoiding the expensive string concatenation operation. We postpone the optimizations however. Inlining `consSC4` into `intP3` yields

```
let intPC4 = fun tD (n,l2) acc ->
   (tD l2) (acc ^ (string_of_int n))
```

That is, the descriptor sequence such as `intPC4 tD` is a function of two arguments, `(n,l2)` and `acc`. To make `(n,l2)` the last argument and bring `l2` in the position suitable for the η-reduction, we flip the order of the arguments `acc` and `(n,l2)`:

```
let intPC5 = fun tD acc (n,l2) ->
  tD (acc ^ (string_of_int n)) l2
```

(we keep in mind that `tD` is the rest of the descriptor sequence, which, after flipping the argument order, too takes `acc` as the first argument). At last, we can curry `intPC5` and η-reduce `l2`:

```
let intPC6 = fun tD acc n ->
  tD (acc ^ (string_of_int n))
```

Analogously, we inline `nilSC4` into the definition of `nilDP` and flip the argument order:

```
let nilDPC6 = fun acc () -> acc
```

Throughout these transformations, `consDP` remains the identity function:

```
let consDPC6 d tD = fun acc -> d tD acc
```

The main function `printf` was defined in Figure 6 to be the identity function. We have changed the `S` sequence to be a difference string, but we still want `printf` to return an ordinary string. We have to change `printf` to pass the empty string as the accumulator. The flip in the argument order makes the accumulator the first argument of the descriptor sequence, so that new `printf` becomes

```
let printfC5 desc args = desc "" args
```

or, η-reduced,

```
let printfC6 desc = desc ""
```

The running example now reads

```
let descPC6 = consDPC6 intPC6
                (consDPC6 (litPC6 "-th character after ")
                  (consDPC6 charPC6
                    (consDPC6 (litPC6 " is ")
                      (consDPC6 charPC6
                        nilDPC6))))
```

whose inferred type is `string -> int -> char -> char -> unit -> string`. Comparison with the type of `descP3` in Figure 6 shows that we have achieved our goal to

Main function:

```
(* val printfC7 : ((’a -> ’a) -> string -> ’b) -> ’b *)
let printfC7 desc = desc (fun x -> x) ""
```

Primitive format descriptors:

```
(* val litPC7 : string -> (string -> ’a) -> string -> ’a *)
let litPC7 str = fun tD acc ->
  tD (acc ^ str)

(* intPC7 : (string -> ’a) -> string -> int -> ’a *)
let intPC7 = fun tD acc n ->
  tD (acc ^ (string_of_int n))

(* charPC7 : (string -> ’a) -> string -> char -> ’a *)
let charPC7 = fun tD acc c ->
  tD (acc ^ (string_of_char c))
```

Infix operator for functional composition:

```
(* val ( ^^ ) : (’a -> ’b) -> (’c -> ’a) -> ’c -> ’b *)
let (^^) f g = fun x -> f (g x)
```

Running example:

```
let descPC7 = intPC7 ^^ (litPC7 "-th character after ") ^^ charPC7 ^^
              (litPC7 " is ") ^^ charPC7
(* val descPC7 : (string -> ’_a) -> string -> int -> char -> char -> ’_a *)

printfC7 descPC7 5 ’a’ ’f’
(*
- : string = "5-th character after a is f"
*)
```

Fig. 7 The polyvariadic context-passing implementation of `printf`.

curry the tupling implementation. The expression `printfC6 descPC6 5 ’a’ ’f’ ()` prints the desired output.

It is inconvenient to have to terminate the argument sequence with the dummy `()`, which corresponds to the empty argument sequence in the tupling implementation. We can trace the dummy `()` to `nilDPC6`, which makes sure that the argument sequence is ‘terminated’. This termination of the argument sequence is not needed: the type of the descriptor sequence, for example, `descPC6`, shows that we cannot obtain the formatting result of the desired type `string` if we pass to `printf` more or fewer arguments than specified in the descriptor. The type checker will point out that error. Therefore, `nilDP` becomes the identity function. Since `consDPC6` is the identity too, we see that the descriptor sequence such as `descPC6` is the functional composition of primitive descriptors applied to `nilDP`. We thus obtain the desired implementation summarized in Figure 7, which is essentially the result by Danvy (1998). The complete derivation is in the file `derive6.ml` in the accompanying code. The string accumulator `acc` is a data-structure continuation (Wand 1980b) that represents a context of the form `consS s1 (... (consS sn []) ...)`.

6 'Direct-style' polyvariadic `printf` implementations

The implementation in Figure 7 fulfills our desiderata for a typed `printf` that takes a variable number of arguments whose types and number match the format descriptor. We have attained our goals, and can stop now. We would like however to get a fuller picture and derive a different group of `printf` implementations. They, too, are polyvariadic and statically assure that the types and the number of the arguments match the format descriptor. The implementations in this section are in a so-called 'direct style', in contrast to the implementation in Figure 7, which is in a context-passing style. The context in question is the context represented by the string accumulator `acc`. Primitive descriptors such as `intPC7` (see Figure 7) take the current value of `acc` as the explicit argument, and pass the augmented `acc` to the rest of the descriptor sequence, represented by `tD`. (Since `acc` can be regarded as a data-structure continuation (Wand 1980b), the implementation in Figure 7 can be regarded as a CPS implementation.)

At the end of the present section, we derive an implementation, Figure 9, in which the primitive descriptors look like

```
let litP7 str = fun () -> str

let intP7 = fun () ->
  let n = shift (fun k -> k) in
  string_of_int n
```

with no accumulator to pass around. The rest of the descriptor sequence is implicit too in the evaluation context of `intP7` rather than being given as the explicit argument `tD`. Such an implementation, aptly called 'direct style', is more concise, as is generally the case (Asai 2009). The elegance of direct-style implementations is one reason to consider them. Direct-style implementations have less 'bureaucracy' – auxiliary arguments such as `acc` to be passed around. Therefore, direct-style implementations are arguably (Asai 2009) easier to use.

Finally, direct-style implementations are insightful because of a duality between direct and continuation-passing styles. In CPS, the 'consumer' of `intPC7`'s formatting result is `acc`, passed explicitly to `intPC7` as an argument. In direct style, the consumer of `intP7`'s result is the implicit context of its invocation. The descriptor `intP7` uses *control operators* such as `shift` above to turn the implicit context into an explicit function. This transformation, *reification*, has been introduced by Friedman and Wand (1984). The correspondence between CPS and direct style, *in the typed setting*, was first studied by Meyer and Wand (1985); Wand (1985).

We derive several direct-style implementations of `printf`; while some of them are known and have been described elsewhere (Asai 2009), the others are novel. In particular, the implementations using so-called multi-prompt delimited control are new and surprising. All in all, we derive four implementations, which can be arranged in two groups. The implementations of the first group require applying `printf` to the values to format inside a so-called 'reset'. That requirement is an imposition, overcome in the implementations of the second group, in §6.3, which are derived by considering the argument sequence to be a context rather than a data structure. Within each group, we show one implementation using the delimited continuation operators shift/reset and one using multi-prompt shift/reset. Only the implementations using multi-prompt shift/reset can be written in OCaml as it is.

Our starting point, as in §5.2, is the refined tupling implementation in Figure 6. Since we want to move away from passing to `printf` the values to format as a nested tuple, we try a different representation for the argument sequence, the one we used for `scanf`, in Figure 5. We recall this representation below:

```
let nilA = fun f -> f
let consA h t = fun f -> t (f h)
```

The sequence `[1; 2]`$_A$ is concretely represented as `fun f -> f 1 2`. To use this representation in `printf`, we need to define the deconstructors of the sequence, in particular, `un_consA`. The deconstructor should satisfy the property that for all `h` and `t` such that `consA h t` is well-typed, `un_consA (consA h t)` is well-typed and evaluates to `(h,t)`. Since `consA h t` is a function, we can only apply it. If we apply the function to some argument `f`, we obtain the value `t (f h)`. We should now choose `f` so to get the desired result `(h,t)`. At first glance, such `f` does not exist. Although the value `h` is passed to `f` as an argument, the value `t` is available only from the context of `f h`. There does not seem to be a way for a function to grab the value from its context. That is where delimited control helps.

6.1 Delimited control

Delimited control is the generalization of exceptions, or, to be precise, of restartable exceptions of Common Lisp. The expression `<e>` (pronounced "reset `e`") is quite like `try e with ex -> ex`: if `e` raises no exceptions during evaluation, both forms, `reset` and `try`, are equivalent to `e`. In particular, for any value `v`, `<v>` evaluates to `v`. Exceptions – or, in general, control effects – are raised by the function `shift`. The function takes one argument, which is a function itself. The application `shift (fun _ -> v)` is equivalent to `raise v`. Therefore,

```
<(1 + shift (fun _ -> 10))>
```

just like

```
try 1 + raise 10 with ex -> ex
```

evaluates to `10`. Like exceptions in Common Lisp, the 'exception' raised by `shift` is restartable. When the application `shift (fun k -> e)` is evaluated, the variable `k` is bound to a function that restarts the exception. Informally, when `k` is applied to a value `v`, the application `shift (fun k -> e)` in the original expression is replaced by `v`, the evaluation continues and the value returned by the closest `< >` block becomes the result of `k v`. For example, `<1 + shift (fun k -> k 10)>` is equivalent to `<let k x = <1 + x> in k 10>` and evaluates to `11`: the 'exception' raised by `shift` is restarted with the value `10`. The example can be re-written as `(<1 + shift (fun k -> k)>) 10`, demonstrating that the restart function, bound to `k`, can be returned as the value of the `< >` expression and invoked later. The restart function is truly a regular function: it can be returned as a value, passed as argument to other functions, stored in data structures, and can be applied several times. These features distinguish delimited control from Common Lisp exceptions: the latter may be restarted only once and only when control remains within the scope of the exception-handling block. Restarting a shift-raised 'exception' several times lets us emulate non-determinism. We do not use multiple restarts in the present paper however; we do crucially depend on the ability to re-enter

Ordinary delimited control:
`<v>` ≡ `v` for any value `v`
`<C[shift v]>` ≡ `<v (fun x -> <C[x]>)>`

Multi-prompt delimited control:
`<v>`$_p$ ≡ `v` for any value `v`
`<C`$_p$`[shift`$_p$` v]>`$_p$ ≡ `<v (fun x -> <C`$_p$`[x]>`$_p$`)>`$_p$

Fig. 8 Ordinary and multi-prompt delimited control. Context `C[]` is a term with a hole that contains no `< >` around the hole. Context `C`$_p$`[]` contains no `< >`$_p$ of the sort p around the hole, but may contain `< >`$_{p'}$ of other sorts p' around the hole.

an exception-handling block after it has finished; we shall see the tell-tale pattern `shift (fun k -> k)` many times. More formally the semantics of delimited control is described in Figure 8. The figure also shows the generalization of delimited control to 'exceptions' of several sorts (identified by so-called *prompts* p): an exception raised by `shift`$_p$ `v` is caught by the closest enclosing `< >`$_p$ of the same sort p (Dybvig et al. 2007; Gunter et al. 1995).

We can now define `un_consA`. Recall that our goal is to find a function `f` such that the application `((fun f -> t (f h)) f)` would evaluate to a pair `(h,t)` for any `h` and `t` of appropriate types. The difficulty was obtaining `t` from the context of the application `f h`. By specializing the equations in Figure 8 we obtain `<t (shift v)>` is equivalent to `<v (fun x -> <t x>)>` for some `v`. Thus `shift` resolved the difficulty, obtaining the needed `t` (or, `(fun x -> <t x>)`, which is almost the same) from the context of the application of `shift v` and giving it to `v` as the argument. What remains is to choose the hitherto 'free' `v` so to satisfy the property of `un_consA`. This leads to the following definition:

```
un_consA x = <x (fun h -> shift (fun t -> (h,t)))>
```

It is straightforward to verify that `un_consA (consA h t)` is equal to `(h,fun x -> <t x>)`. Although the result differs from the desired `(h,t)`, the difference is, fortunately, insignificant. The value `t` is of the type sequence; in fact, it is the tail of the sequence `consA h t`. The only operations on an argument sequence within `printf` are passing the sequence to the functions `is_nilA` and `un_consA`. (We could have used the ML module system and abstract types to ensure that the values constructed by `consA` and `nilA` could be analyzed only by `un_consA` and `is_nilA`, or passed around.) As we just saw, `un_consA t` is `<t (fun h -> ...)>`. It is easy to conclude from Figure 8 that `< >` is idempotent, and so `un_consA t` is equivalent to `un_consA (fun x -> <t x>)`.

It only remains to define `is_nilA`, which should satisfy the property that `is_nilA x` terminates if and only if `x` is the empty sequence, that is, `fun x -> x`. The following weaker check is sufficient.

```
let is_nilA x = x ()
```

Indeed, if `x` is the identity, `is_nilA x` obviously terminates and returns the desired result `()`. If `x` is a non-empty sequence (that is, has the form `fun f -> f arg1 ...`), then `is_nilA x` is ill-typed.

Substituting thus defined `un_consA` and `is_nilA` into Figure 6 gives us a new implementation of `printf`. We obtain

`printf [intP4; litP4 "-th character after "; charP4; litP4 " is "; charP4]`$_D$
        `[5; 'a'; 'f']`$_A$
`= "5-th character after a is f"`

That is still not quite what we want, because `printf` is not a polyvariadic function. It receives, besides the descriptor, just one argument: `[5; 'a'; 'f']`$_A$, which is a function `fun f -> f 5 'a' 'f'`. Again delimited continuations help. Let us see if we can define `printf` in such a way that the polyvariadic `printf desc arg1 ...` becomes equivalent to our current formulation `desc (fun f -> f arg1 ...)`. That is possible if we enclose the whole `printf` expression in `reset` and observe that `<shift v arg1 ...>` is equivalent to `<v (fun x -> <x arg1 ...>)>`. We again appeal to the fact that if all the uses of the argument sequence `t` is passing it to `un_consA` and `is_nilA`, then `t` and `fun x -> <t x>` are equivalent. We thus obtain

```
let printf desc = shift desc
```

and so

```
let descP4 = [intP4; litP4 "-th character after "; charP4;
              litP4 " is "; charP4]D
in <printf descP4 5 'a' 'f'>
```

evaluates to the desired string `"5-th character after a is f"`. We have just obtained a novel implementation of `printf` using delimited control. The implementation is typed, as we see next. A crucial step in the derivation was reification (Friedman and Wand 1984): we reified the *context* of the `printf` application, `[] arg1 arg2 ...`, into a term, a function `fun f -> f arg1 arg2 ...`.

6.2 Types of delimited control

We have so far ignored the typing of delimited control, which is a rather delicate matter. Meyer and Wand (1985); Wand (1985) gave the first typed treatment of continuations. Danvy and Filinski (1989) then developed it further for shift/reset, and Asai and Kameyama (2007) later generalized it to polymorphic shift/reset. Since we are implementing `printf` in the typed language OCaml, the typing is of immediate interest.

To type an expression involving shift/reset, we need to keep track of not only the type of the expression but also the *answer type* of its context. We describe the answer type on the example of `<if shift (fun k -> M) then 1 else 2>` where `M` is some term. To type `(fun k -> M)` in this expression, we need to know the type of `k`. Since `shift (fun k -> M)` appears in the context of the test expression of the `if` statement, the shift expression must have the type `bool`. Hence, if the 'exception' raised by `shift` here is to be restarted, it must be resumed with a boolean value. Therefore, `k` is a function whose argument type is `bool`. To find the result type of `k`, we should look farther – to the whole expression enclosed by reset, which is the whole `if` expression in our case. The type of the expression is `int`. Thus `k` has the type `bool -> int`. The type of the surrounding reset expression, i.e., `int` in this case, is called the answer type. Therefore, we can successfully type the expression `<if shift (fun k -> k false) then 1 else 2>`, which is equivalent to `<let k x = <if x then 1 else 2> in k false>`.

Suppose however that `M` in the above example is `string_of_int (1 + (k false))`. According to Figure 8, the whole reset-expression is equivalent to `<let k x = <if x then 1 else 2> in <string_of_int (1 + (k false))>>` and evaluates to the string value `"3"`. Although the captured continuation `k` returns a value of type `int`, the whole expression returns a result of type `string`. That is, the answer type has been changed.

There is nothing wrong with this answer-type modification. Although the context had originally `int` as its answer type, it was captured as a composable function, bound to `k`, and removed from the current context completely. The answer type `int` is then just the type of the codomain of `k`, unrelated to the answer type where `k` is invoked. The answer type where `k` is invoked is determined by `M` only, and it need not be the same as before. Thus, to type expressions with shift/reset, we need to keep track of an answer type as well as how it is modified by shift.

We need the answer-type modification to type `un_consA`. Indeed, the type of `consA` is `'a -> ('b -> 'c) -> ('a -> 'b) -> 'c`. Therefore, in an expression `<(consA h t) (fun h -> shift (fun t -> (h,t)))>` the application of `consA h t` has the type `'c` so that `t` has the type `'b -> 'c`. However, the control effect of `shift` changes the type of the whole expression from `'c` to `'a * ('b -> 'c)`, the type of the pair `(h,t)`.

Implementing `shift/reset` with the answer-type modification is difficult within the existing ML system. Essentially, we need effect typing: we need to assign an expression not only the type of its value, but also the type of its effect (the answer type and its change). Incorporating such effect types is a non-trivial change to the existing type checker. For example, to implement shift/reset in the MinCaml compiler, Masuko and Asai (2009) had to replace the type system completely to incorporate answer types. Although it is a straightforward extension of Hindley-Milner type system, how to incorporate more sophisticated constructs such as modules is not clear. We can however embed the calculus of Asai and Kameyama (2007) in Haskell, using so-called parameterized monads. The embedding lets us implement `un_consA` and `printf` of §6.1, which we demonstrate in the file `Derive5.hs` of the accompanying code.

Less intrusive to typing are so-called multi-prompt delimited control operators, mentioned in Figure 8 (Dybvig et al. 2007; Gunter et al. 1995). They have been implemented in OCaml (Kiselyov 2010). The implementation provides so-called prompts, which are typed values to mark the 'flavor' of `reset` and `shift`. The control operation $\texttt{shift}_p$ `e` is written in code as `shift p e`, where `p` is the prompt; $\texttt{<e>}_p$ is written as `push_prompt p (fun () -> e)`. New prompts are created by the operation `new_prompt ()`. The advantage of introducing explicit prompts is the straightforward typing of the control operators:

```
val new_prompt   : unit -> 'a prompt
val shift        : 'a prompt -> (('b -> 'a) -> 'a) -> 'b
val push_prompt  : 'a prompt -> (unit -> 'a) -> 'a
```

The typing is a bit 'simplistic' as noted already in (Gunter et al. 1995): a well-typed code can still attempt to evaluate $\texttt{shift}_p$ `e` outside any enclosing $\texttt{<·>}_p$, which leads to a run-time error, quite like an 'uncaught exception'. In contrast, the system of Danvy and Filinski (1989) for single-prompt shift/reset prevents such run-time errors, by simply enclosing the whole expression into `<·>`.

A prompt value has the type `'a prompt` where `'a` is the answer type. Clearly the system supports no answer-type polymorphism or answer-type modification: all control operations dealing with the prompt `p` have the same answer type, included in the type of `p`. In particular, in the type of `shift` above, the type `'a` appears both as the type of the codomain of the captured continuation and as the final answer type. The question emerges if `un_consA` and the `printf` implementation of §6.1 can at all be written in OCaml. The answer turns out affirmative: in some situations, the ability to create arbitrarily many typed prompts makes up for the lack of answer-type modification: the quantity of prompts makes up for their quality.

The OCaml definition for `un_consA` is as follows

```
(* val un_consA : (('a -> 'b) -> 'c) -> 'a * ('b -> 'c) *)
let un_consA x =
   let pr = new_prompt () in
   let p = new_prompt () in
   push_prompt pr (fun () ->
     push_prompt p (fun () ->
       x (fun h -> shift p (fun t -> abort pr (h,t))));
     failwith "unreachable")
```

The inferred type (shown in the comments) confirms its operation. Here

```
let abort p v = shift p (fun _ -> v)
```

raises the 'exception' v to be caught by the closest enclosing $<\cdot>_p$.

If we disregard `abort pr` and `failwith` for a moment, the definition looks the same as `un_consA` in §6.1. As we explained earlier, the application `x (fun h -> ...)` has the type `'c` – which is the type of the `push_prompt p (fun () -> ...)` expression – the answer type associated with the prompt `p`. However, we want to return `(h,t)`, which is of a different type `'a * ('b -> 'c)`. Our only choice to communicate this value is to 'throw' it as an exception, to be caught by the enclosing `push_prompt pr (fun () -> ...)`. Since the expression in the body of the latter `push_prompt` returns by `abort`ing rather than normally, we can insert the `failwith` expression. The latter expression never returns, and so can be given an arbitrary type. The net effect is the 'change' of the type of the body from `'c` (the type of `push_prompt p (fun () -> ...)`) to the type `'a * ('b -> 'c)`. The demonstrated pattern – aborting to an auxiliary prompt `pr` and using `failwith` to 'change' the type – appears quite general; in particular, it was used by Kiselyov et al. (2006; Sec 5.2).

The appearances of `failwith` and `abort` remind us that the correctness of this emulation of the answer-type modifying delimited control has to be shown separately, as we have just outlined. Previously we have relied on the type system for justification of our derivations, for instance, of `un_consA` of §6.1. To prove that for all `h` and `t`, `fst (un_consA (consA h t))` is equal to `h` we merely had to look at the type of that expression in the calculus of Asai and Kameyama (2007) (keeping in mind that the calculus is strongly normalizing and relying on parametricity). We can no longer justify the same way the correctness of `un_consA` written with multi-prompt delimited control; a more elaborate argument is required. Once it is shown that `un_consA` satisfies the property of the sequence deconstructor, using `un_consA` gives the typed `printf` with the same static assurances as the other implementations. The accompanying code presents the complete OCaml implementation for `printf` at the end of §6.1. It has not been known before.

6.3 Deriving polyvariadic `printf`

We have almost achieved a polyvariadic `printf` that accepts the values to format as function arguments. However, we have to enclose the whole `printf` expression in reset.

Here we derive a different implementation of direct-style polyvariadic `printf`, which corresponds to the context-passing implementation of §5.2. Unlike the latter, we eschew the conversion to the accumulator-passing style and flipping of the argument order. We start again with the refined tupling implementation in Figure 6 and adjust it by replacing the nested tuple implementation of the argument sequence with a different one. In other words, we choose the deconstructors `un_consA` and `is_nilA` so to obtain the desired `printf`. We desire `printf` that can be used like `D[printf (consDP intP (consDP charP ...)) 1 'c' ...]` where `D[]` is the context of the whole program. Since `intP` includes `un_consA` (see Figure 6), the above expression reduces to `D[ C[un_consA arg] 1 'c' ...]` where `C[]` is some evaluation context and `arg` stands for a value being passed to `un_consA`. Recall that one may view `un_consA arg` as 'reading' from a stream described by a 'handle' `arg`. Here, the stream is the sequence of arguments applied to `printf`. The argument sequence received by `printf` turns out to be a *context* rather than a term. Now, we would like the above expression to reduce to `D[ C[(1,arg)] 'c' ...]`, the result of 'reading' the argument `1`. We obtain the specification for the desired `un_consA`: `C[un_consA arg]` should reduce to `fun y -> C[(y,arg)]`. Examining the reduction rules for `shift` in Figure 8 shows a way to satisfy the specification, if we assume that the context `C[]` is enclosed in a reset. The assumption is easy to satisfy by defining the main function `printf` to introduce the required reset around the descriptor sequence. We thus arrive at the following deconstructors:

```
is_nilA1 arg  = ()
un_consA1 arg = shift (fun k -> fun y -> k (y,arg))
```

We can substitute the above definition into `intP3` obtaining

```
let intP5 = fun tD arg ->
  let (n,arg) = shift (fun k -> fun n -> k (n,arg)) in
  consS (string_of_int n) (tD arg)
```

A few straightforward simplifications immediately suggest themselves. First we observe that `arg`, the argument sequence handle, is simply passed around never to be examined. We can set `arg` to be a fixed value, for example, `()`:

```
let intP5 = fun tD () ->
  let n = fst (shift (fun k -> fun n -> k (n,()))) in
  consS (string_of_int n) (tD ())
```

which we simplify further by using a law `fst (shift f)` $\equiv$ `shift (fun k -> f (fun x -> <k (fst x)>))` derivable from the axioms of Kameyama and Hasegawa (2003). The sub-expression `fst (shift (fun k -> fun n -> k (n,())))` thus becomes `shift (fun k -> fun n -> k n)`, which simplifies further by η-reducing the argument of `shift`. We obtain:

```
let intP5 = fun tD () ->
  let n = shift (fun k -> k) in
  consS (string_of_int n) (tD ())
```

The primitive descriptor `litP3` does not invoke `un_consA` and hence remains the same:

```
let litP5 str = fun tD () ->
  consS str (tD ())
```

By inlining `is_nilA1` into `nilDP` of Figure 6, we obtain the expression for the empty descriptor sequence

```
let nilDP5 = fun () -> nilS
```

The descriptor sequence is a function of a dummy argument, a thunk. The main function `printf` needs to force the thunk, not forgetting to introduce reset:

```
let printf desc = <desc ()>
```

We are very close to achieving our desiderata (put forth at the beginning of §6) and deriving the implementation of `printf` that avoids threading through of not only the accumulator but also of the rest of the descriptor sequence. We want the rest of the descriptor sequence to be implicit in the evaluation context rather than being given as the explicit argument `tD`. We need to find a way to get rid of `tD`, which we can do by η-reducing it away. Recalling that `consS` is string concatenation, we can write `intP5` in a form suggestive of further η-reducing the arguments `tD` and `()` away:

```
let intP5 = fun tD () ->
  let n = shift (fun k -> k) in string_of_int n ^ tD ()
```

However, η-reductions in a call-by-value calculus are only sound if the result is a value or a pure (Sabry 1998), assuredly non-divergent expression. Expressions involving `shift` are effectful and certainly not pure. The form of `intP5` above shows two parts: first, obtaining the number `n` and converting it to a string; second, adding the result to the output sequence. The following series of equivalence transformations lets us separate the two parts. Since we are dealing with expressions containing `shift`, we have to be mindful of evaluation order.

The first transformation is to name the result of `string_of_int n`:

```
let intP5 = fun tD () ->
  let n = shift (fun k -> k) in
  let x = string_of_int n in
  x ^ tD ()
```

The transformation is equivalence-preserving regardless of the evaluation order for the expression `string_of_int n ^ tD ()` since `string_of_int n` is a pure expression. We re-associate the let-bindings:

```
let intP5 = fun tD () ->
  let x =
    let n = shift (fun k -> k) in string_of_int n
  in x ^ tD ()
```

and η-expand:

```
let intP5 = fun tD () ->
  let x =
    (fun () -> let n = shift (fun k -> k) in string_of_int n) ()
  in x ^ tD ()
```

The two parts of `intP5` can now be separated:

```
let intP7 = fun () ->
  let n = shift (fun k -> k) in string_of_int n
let (^^^) = fun f g () ->
  let x = f () in x ^ g ()
let intP5 = fun tD () -> (intP7 ^^^ tD) ()
```

Keeping in mind that `intP7 ^^^ tD` is a pure expression (evaluating to a thunk) and so is `(^^^) intP7`, we can finally perform two η-reductions on `intP5`, yielding

Main function:

```
let printf7 desc = <desc ()>
```

Primitive format descriptors:

```
let litP7 str = fun () -> str

let intP7 = fun () ->
  let n = shift (fun k -> k) in
  string_of_int n

let charP7 = fun () ->
  let c = shift (fun k -> k) in
  string_of_char c
```

Infix operator for descriptor composition:

```
let (^^^) = fun f g () -> let x = f () in x ^ g ()
```

Running example:

```
let descP7 = intP7 ^^^ (litP7 "-th character after ") ^^^ charP7 ^^^
             (litP7 " is ") ^^^ charP7

printf7 descP7 5 'a' 'f'
(*
- : string = "5-th character after a is f"
*)
```

Fig. 9 The polyvariadic direct implementation of `printf`.

```
let intP5 = (^^^) intP7
```

The builder of the descriptor sequence `consP` remains the identity. The observation made at the end of §5.2 still applies: the descriptor sequence is a functional composition of primitive descriptors applied to `nilDP5`. The operation `^^^` is string concatenation lifted to thunks producing strings. It is an associative operation, and `nilDP5` is its neutral element. This observation leads to the final implementation in Figure 9. If we assume left-to-right evaluation, we can further reduce our implementation to coincide with the simplest implementation of the polyvariadic `printf` in Asai (2009).

To type the implementation we need the type system for shift/reset that supports answer-type modification (Asai 2009) and answer-type polymorphism (Asai and Kameyama 2007). Therefore, we cannot type the implementation in Figure 9 in OCaml. The accompanying code, `Derive5.hs`, near the end shows the complete implementation of Figure 9 using the emulation of Asai and Kameyama's calculus in Haskell.

Surprisingly, we can also use multi-prompt delimited control, which are available in OCaml. Once again, the quantity of prompts makes up for their quality (the fixed, monomorphic and unmodifiable answer type). The complete OCaml implementation is available at the end of `derive5.ml`.

7 Related work

Danvy's surprising discovery (Danvy 1998) that statically safe `printf` can, after all, be typed in the Hindley-Milner type system has created the field of dependent-type–like programming and stimulated the search for further properties that can be statically assured in simpler-typed systems.

Following Danvy's work, Asai (2009) presented three solutions to the `printf` problem. His solutions are essentially obtained by transforming Danvy's CPS-based solution back to direct style using delimited-control operators. The simplest solution uses literal strings as format descriptors and string concatenation for their composition. To type this solution, however, requires the type system for shift/reset with answer-type modification and polymorphism. In the current paper, we developed essentially the same solution (among many others) from the specification in a rigorous manner.

Hinze (2003) presented a general solution to the `printf` problem in Haskell. His solution too is in direct style, and so has to be able to express the answer type and its modification by format descriptors. In his system, the answer-type modification is expressed as a type-level function – or, to be precise, as a type that can be interpreted as a type-level function. The primitive descriptor `lit x` is associated with a type that can be interpreted as the identity function; the primitive descriptor `int` is associated with the type that can be interpreted as a function transforming a type `t` to type `Int -> t`. The format sequence constructor (`consD` in our terminology) receives the type that can be interpreted as a composition of (type) functions. Hinze uses Haskell type classes to interpret a type code as a type function and compute the result of applying the function to a given type.

For a long time, computational linguists and programming-language researchers have sought to use the same grammar representation for both parsing and generation (Kay 1975; Rendel and Ostermann 2010; Shieber 1988). A view of the type-safe formatted-IO library as multiple interpretations of the DSL of format descriptors is expounded in `http://okmij.org/ftp/typed-formatting/`. That web page presents two ways to encode the DSL in Haskell: as an initial algebra, using a GADT, or as a final algebra, using a type class. Although one may translate these implementations into OCaml, the result is ungainly.

8 Conclusions

We have presented a general specification of formatted IO as a `zipWith` procedure for heterogeneous sequences of format descriptors, values to format, and strings. We have *systematically* derived several type-safe implementations of the specification in OCaml. Some of our implementations closely resemble OCaml's built-in formatted-IO facility; our implementations require no ad-hoc extensions to the Hindley-Milner type system and therefore are portable and extensible. Our exposition is the first systematic exploration of the design space of typed formatted IO in Hindley-Milner type systems. Some of the derived implementations have not been known or anticipated before.

Writing the paper has been like writing a biographical novel. Most of the 'facts' – implementations of `printf` and `scanf` – are already known. As we struggled to come up with a good story behind the events, we discovered several new facts. Our guiding principles were (i) to specify modules by sentences containing the names of functions and constructors as non-logical symbols; (ii) to represent a DSL as a final algebra; (iii) to transform programs by fusing a term with a part of its context. These principles are due to Mitchell Wand.

Acknowledgements We would like to thank Yukiyoshi Kameyama and Mitch Wand for helpful discussions, and Olivier Danvy for many comments and encouragement. Many improvements to the presentation by the anonymous reviewers are gratefully acknowledged.

References

Asai, Kenichi. 2009. On typing delimited continuations: Three new solutions to the printf problem. *Higher-Order and Symbolic Computation* 22(3):275–291.

Asai, Kenichi, and Yukiyoshi Kameyama. 2007. Polymorphic delimited continuations. In *Proceedings of APLAS 2007: 5th Asian symposium on programming languages and systems*, ed. Zhong Shao, 239–254. Lecture Notes in Computer Science 4807, Berlin: Springer.

Carette, Jacques, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19(5):509–543.

Danvy, Olivier. 1998. Functional unparsing. *Journal of Functional Programming* 8(6): 621–625.

Danvy, Olivier, and Andrzej Filinski. 1989. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark. `http://www.cs.au.dk/~danvy/Papers/fatc.ps.gz`.

Dybvig, R. Kent, Simon L. Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of Functional Programming* 17(6):687–730.

Felleisen, Matthias, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. 1988. Abstract continuations: A mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM conference on Lisp and functional programming*, 52–62. New York: ACM Press.

Friedman, Daniel P., and Mitchell Wand. 1984. Reification: Reflection without metaphysics. In LFP (1984), 348–355.

Friedman, Daniel P., and Mitchell Wand. 2008. *Essentials of programming languages.* 3rd ed. Cambridge: MIT Press.

Goguen, J. A., J. W. Thatcher, and E. G. Wagner. 1978. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current trends in programming methodology*, ed. Raymond T. Yeh, vol. 4, 80–149. Englewood Cliffs, NJ: Prentice-Hall.

Gunter, Carl A., Didier Rémy, and Jon G. Riecke. 1995. A generalization of exceptions and control in ML-like languages. In *Functional programming languages and computer architecture: 7th conference*, ed. Simon L. Peyton Jones, 12–23. New York: ACM Press.

Haynes, Christopher T., Daniel P. Friedman, and Mitchell Wand. 1984. Continuations and coroutines. In LFP (1984), 293–298.

Hinze, Ralf. 2003. Formatting: a class act. *Journal of Functional Programming* 13(5): 935–944.

Hudak, Paul. 1996. Building domain-specific embedded languages. *ACM Computing Surveys* 28(4es):196.

Hutton, Graham. 1999. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming* 9(4):355–372.

Kameyama, Yukiyoshi, and Masahito Hasegawa. 2003. A sound and complete axiomatization of delimited continuations. In *ICFP '03: Proceedings of the ACM international conference on functional programming*, 177–188. New York: ACM Press.

Kamin, Samuel. 1983. Final data types and their specification. *ACM Transactions on Programming Languages and Systems* 5(1):97–121.

Kay, Martin. 1975. Syntactic processing and functional sentence perspective. In *TINLAP '75: Proceedings of the 1975 workshop on theoretical issues in natural language* *processing*, 12–15. Association for Computational Linguistics.

Kiselyov, Oleg. 2010. Delimited control in OCaml, abstractly and concretely: System description. In *Proc. FLOPS 2010: 10th international symposium on functional and logic programming*, 304–320. Lecture Notes in Computer Science 6009, Berlin: Springer.

Kiselyov, Oleg, Chung-chieh Shan, and Amr Sabry. 2006. Delimited dynamic binding. In *ICFP '06: Proceedings of the ACM international conference on functional programming*, 26–37. New York: ACM Press.

Kohlbecker, Eugene E., and Mitchell Wand. 1987. Macro-by-example: Deriving syntactic transformations from their specifications. In *POPL '87: Conference record of the annual ACM symposium on principles of programming languages*, 77–84. New York: ACM Press.

LFP. 1984. *Proceedings of the 1984 ACM symposium on Lisp and functional programming*. New York: ACM Press.

Masuko, Moe, and Kenichi Asai. 2009. Direct implementation of shift and reset in the MinCaml compiler. In *ACM SIGPLAN Workshop on ML*, 49–60. New York: ACM Press.

Meyer, Albert R., and Mitchell Wand. 1985. Continuation semantics in typed lambda-calculi (summary). In *Logics of programs*, ed. Rohit Parikh, 219–224. Lecture Notes in Computer Science 193, Berlin: Springer.

Rendel, Tillmann, and Klaus Ostermann. 2010. Invertible syntax descriptions: Unifying parsing and pretty printing. In *Proceedings of the 3rd ACM SIGPLAN symposium on Haskell*, ed. Jeremy Gibbons, 1–12. New York: ACM Press.

Sabry, Amr. 1998. What is a purely functional language? *Journal of Functional Programming* 8(1):1–22.

Shieber, Stuart M. 1988. A uniform architecture for parsing and generation. In *COLING '88: Proceedings of the 12th international conference on computational linguistics*, vol. 2, 614–619.

Swierstra, S. Doaitse. 2009. Combinator parsers: a short tutorial. In *Language engineering and rigorous software development*, ed. A. Bove, L. Barbosa, A. Pardo, and J. Sousa Pinto, vol. 5520 of *Lecture Notes in Computer Science*, 252–300. Spinger.

Wadler, Philip L. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science* 73(2):231–248.

Wand, Mitchell. 1979. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences* 19(1):27–44.

Wand, Mitchell. 1980a. Continuation-based multiprocessing. In *Proceedings of the 1980 ACM conference on Lisp and functional programming*, 19–28. New York: ACM Press. Reprinted in Higher-Order and Symbolic Computation 12(3):285–299, 1999, with a foreword (**?**).

Wand, Mitchell. 1980b. Continuation-based program transformation strategies. *Journal of the ACM* 27(1):164–180.

Wand, Mitchell. 1982a. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems* 4(3):496–517.

Wand, Mitchell. 1982b. Specifications, models, and implementations of data abstractions. *Theoretical Computer Science* 20(1):3–32.

Wand, Mitchell. 1985. Embedding type structure in semantics. In *POPL '85: Conference record of the annual ACM symposium on principles of programming languages*, 1–6. New York: ACM Press.

Xi, Hongwei, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 224–235. New York: ACM Press.